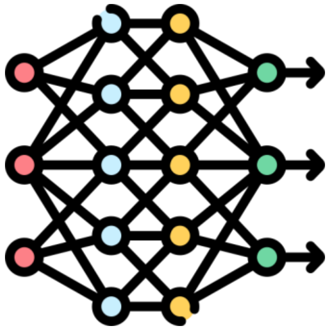
Optimizing Deep Learning Inference via Global Analysis and Tensor Expressions

Chunwei Xia[1,2], Jiacheng Zhao[2] ✉, Qianqi Sun[2], Zheng Wang[1], Yuan Wen[3], Teng Yu[4], Xiaobing Feng[2], Huimin Cui[2]

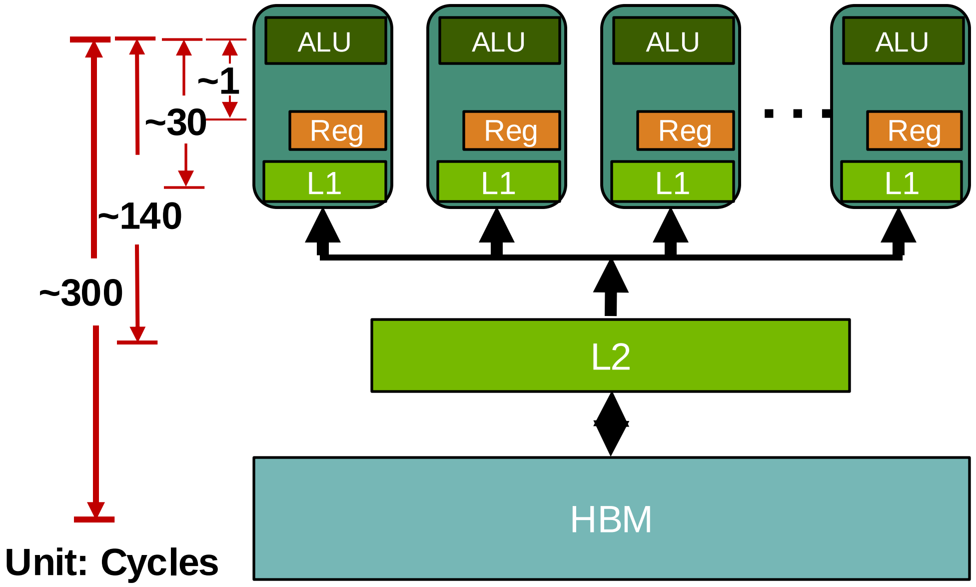
1 School of Computing, University of Leeds
2 SKLP, Institute of Computing Technology, CAS
3 University of Aberdeen
4 THeWake Research

Deep Learning compilers

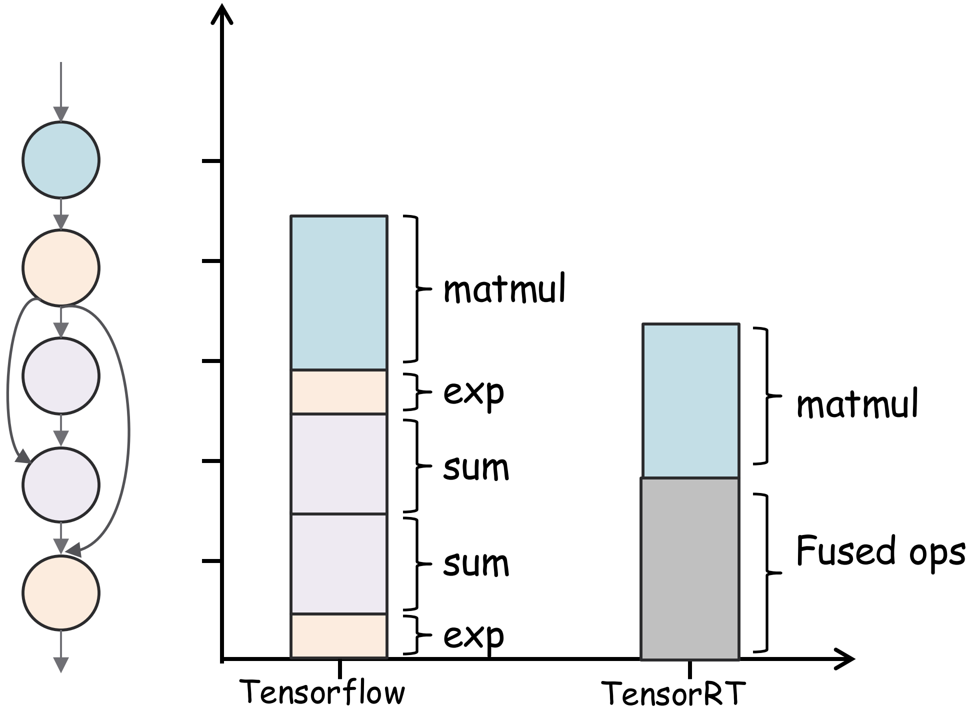


```
__global__ void saxpy(int n,
float a, float *x, float *y) {
int i = blockIdx.x*blockDim.x +
threadIdx.x;
 if (i < n) y[i] = a*x[i] + y[i];
...
}
```

Operator fusion



Memory hierarchy with access latency



DL compiler can reduce inference latency by Operator fusion

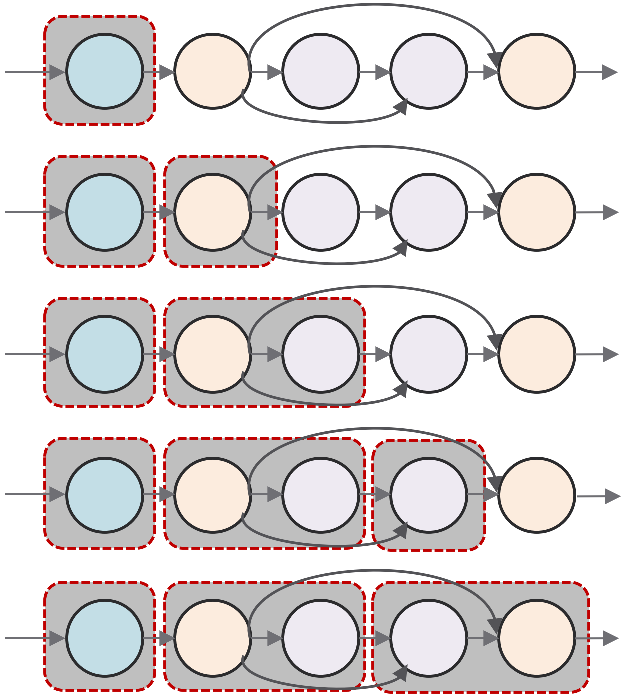
State-of-the-art

- [SystemML, VLDB’18]
- [DNNFusion, PLDI’21]
- [DLFusion, ISPA’20]
- [Apollo, MLSys’22]
- [Rammer, OSDI’20]
- [TASO, OSDI’20],
- [HFUSE, CGO’21]

Representative operator	Second op / First op	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
Add, Relu	One-to-One	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
Expand	One-to-Many	One-to-Many	One-to-Many	×	One-to-Many	One-to-Many
Conv, GEMM	Many-to-Many	Many-to-Many	Many-to-Many	×	Many-to-Many	Many-to-Many
Reshape	Reorganize	Reorganize	One-to-Many	Many-to-Many	Reorganize	Reorganize
Transpose	Shuffle	Shuffle	One-to-Many	Many-to-Many	Reorganize	Shuffle

DNNFusion: Rule base operator fusion

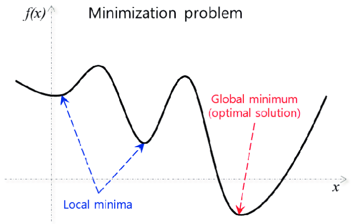
State-of-the-art



3 kernels

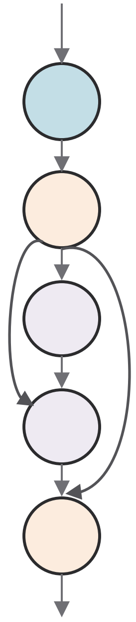
Fuse Ops
Then
Codegen

1. Rule/Heuristic based → Bad extensibility
2. Local optimization →Bad data reuse

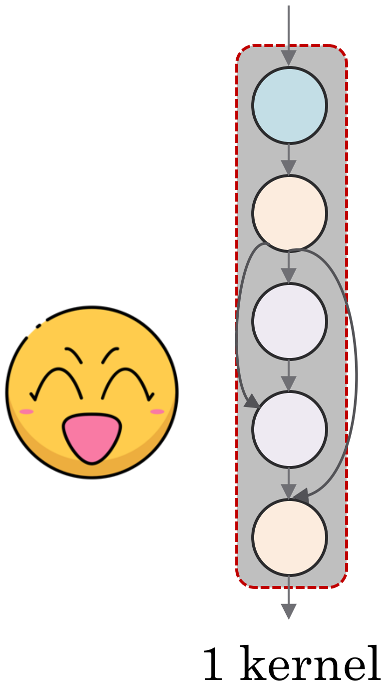


Bottom-Up Operator fusion

What we need

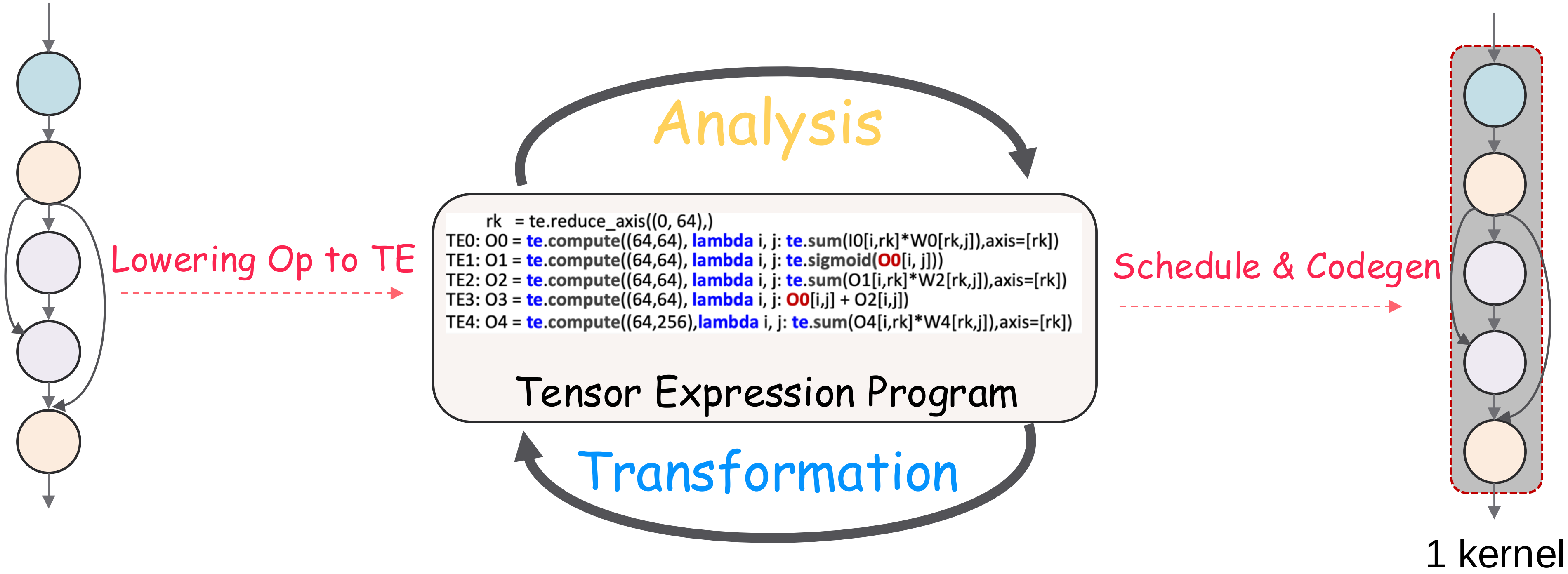


1. Maximize data reuse
2. Fit onto hardware
3. Fully automated



Ultimate Goal: One model to a single kernel

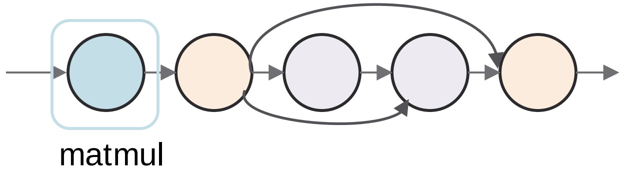
Our approach



- Tensor Expression As the Intermedia Representation
- Try to generate the whole model as a single kernel

Top-Down Global Opt.

Tensor Expression



Lowering Op to TE

```
# Matmul TVM Tensor Expression
def matmul(n, k, m):
    rik = te.reduce_axis((0, k), name='rik')
    A = te.placeholder((m, k), name='A')
    B = te.placeholder((k, n), name='B')
    C = te.compute((m, n),
        lambda i, j: te.sum(A[i, rik] * B[rik, j], axis=[rik]))
    return A, B, C, rik
```

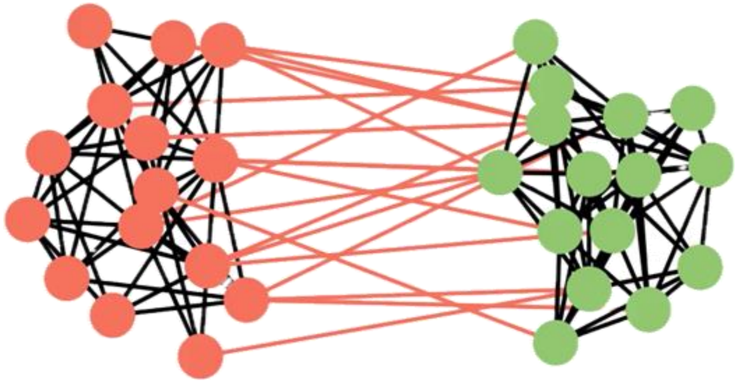
Reduction axis with range (0, k)

I/O Tensor shape

Computation rule

Parallel
Iteration var

Our approach



$$TA = B$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Global Analysis → Graph Partitioning → Automatic transformation

Our approach

$$TA=B$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Global Analysis → Graph Partitioning → Automatic transformation

Our approach

Global Analysis

TE

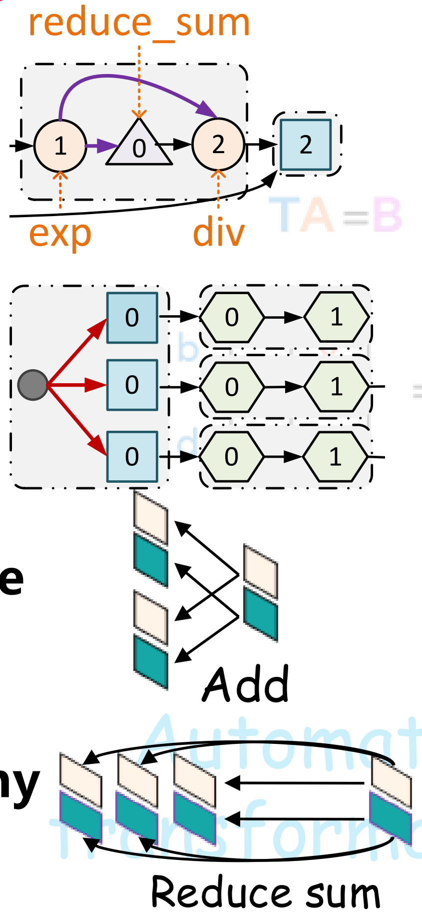
(Compute-intensity, opt schedule)

coarse-grained Tensor (Data reuse)

- **Temporal reuse**
- **Spatial reuse**

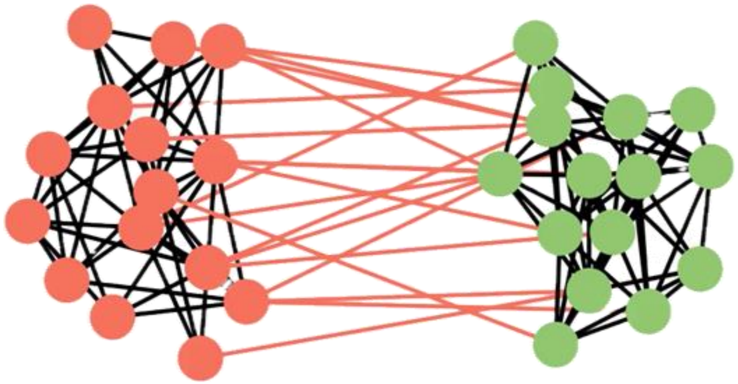
Fine-grained Element (Dependency)

- **One-relies-one-one**
- **One-relies-on-many**



Our approach

Global Analysis



Graph Partitioning

Automatic transformation

Our approach

GPU has limited resources

```
<blockDim(2), threadDim(128)>kernel1(float* a,...){
__shared__ char shared_pool [4*1024*1024];
//gemm1 code
...
}
```

```
<blockDim(8), threadDim(128)>kernel2(float* a,...){
__shared__ char shared_pool [1*1024*1024];
//gemm2 code
...
}
```

```
...
```

```
kernelN
```

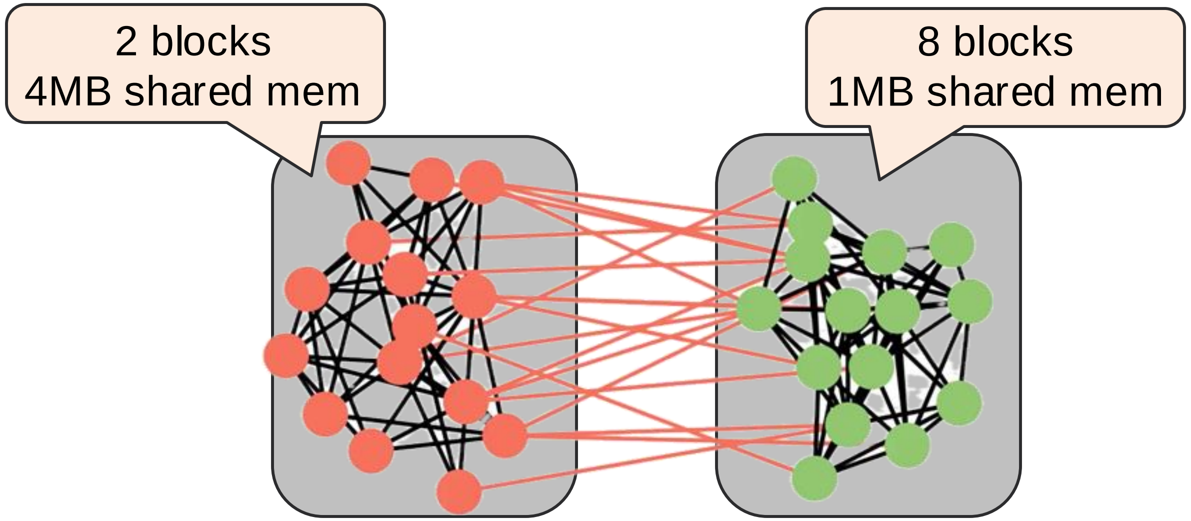
Block Count limit

Shared memory limit

```
<blockDim(8), threadDim(128)>fused(float* a,...){
__shared__ char shared_pool [4*1024*1024];
//kernel1 code
...
Grid.sync(); //Global synch
//kernel2 code
...
Grid.sync(); //Global synch
kerneln
}
```

One model to even a single kernel

Our approach



$TA = B$

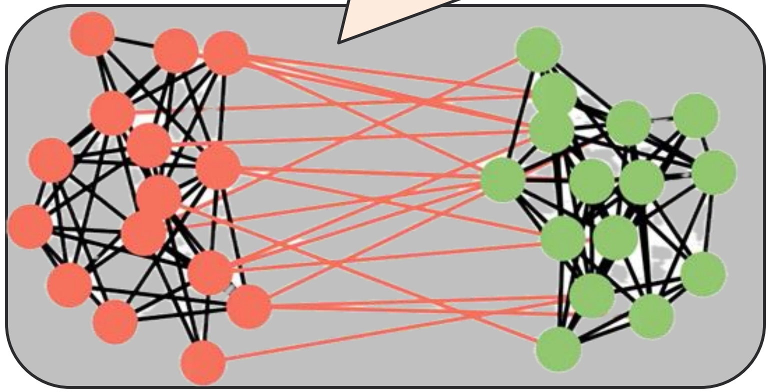
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Resource constraint:
8MB shared memory

Global Analysis → **Graph Partitioning** → Automatic transformation

Our approach

Max(2,8)=8 blocks
Max(1,4)=4MB shared mem



Resource constraint:
8MB shared memory

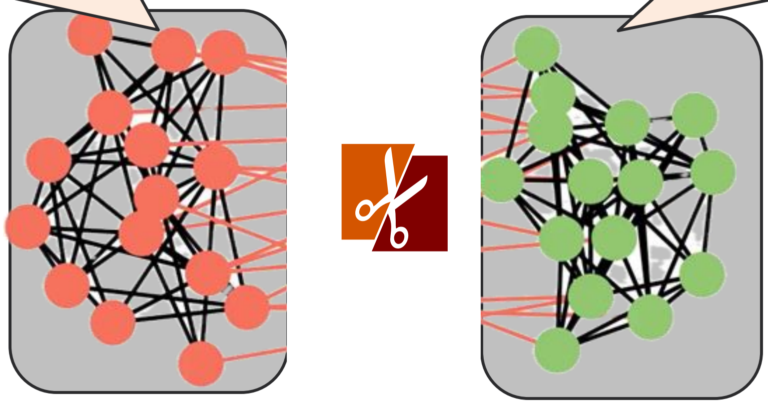
$8 \times 4 = 32 > 8$
Can't fit into the hardware,
Split!

Global Analysis → Graph Partitioning → Automatic transformation

Our approach

TE subprogram
Basic unit for TE transformation

TE subprogram
Basic unit for TE transformation



TA=B

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Candidate Partitioning points:
Compute-intensive TEs

Global Analysis → Graph Partitioning → Automatic transformation

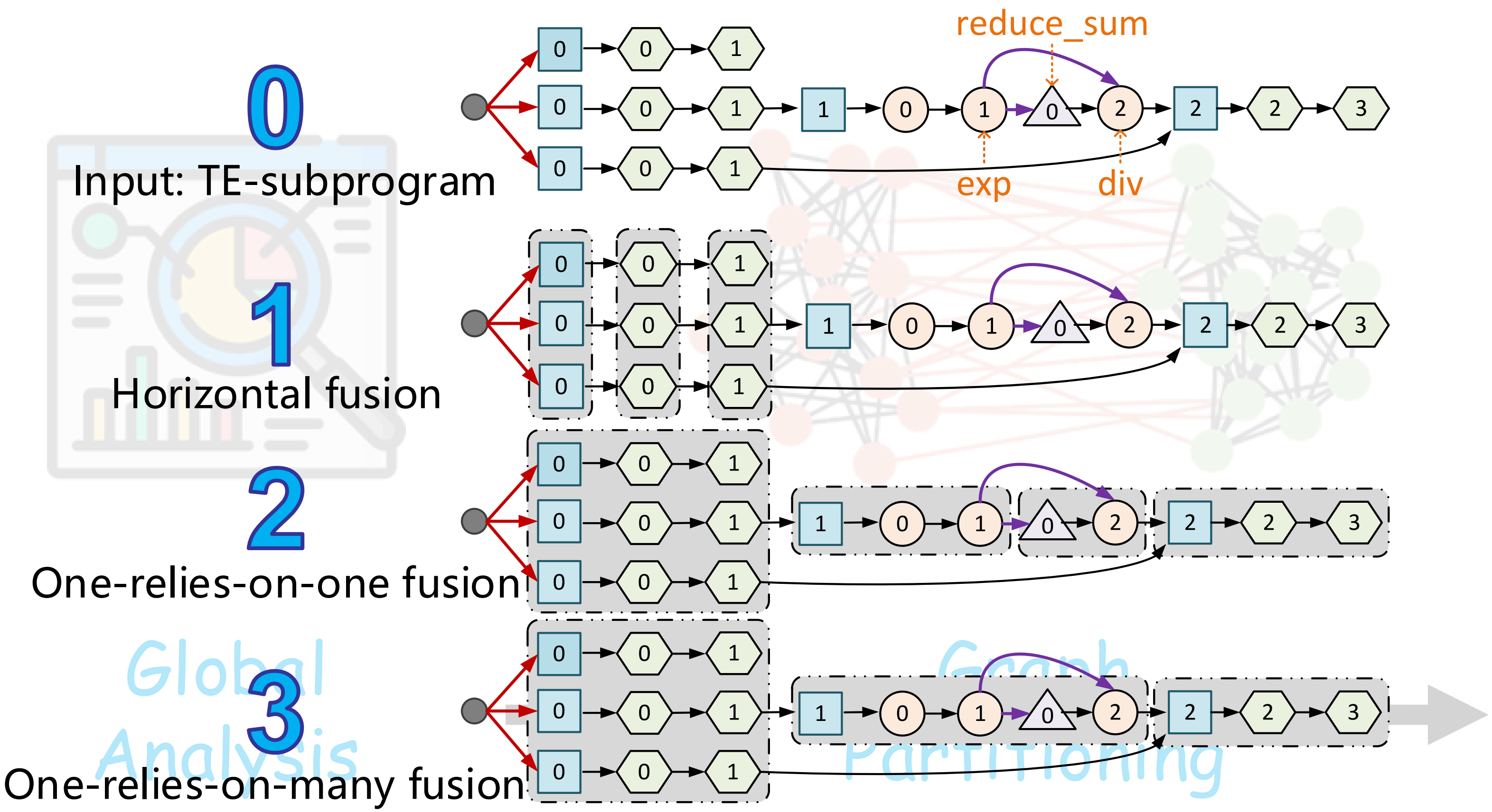
Our approach

$TA = B$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Global Analysis → Graph Partitioning → Automatic transformation

Our approach



$TA = B$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Automatic transformation

Our approach

Before transformation

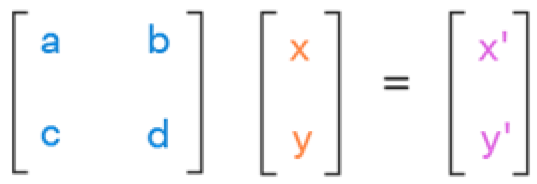
```
A = te.placeholder((4, 8))
B = te.compute((4,8),lambda i,j:
  tir.if_then_else(A[i,j]>0, A[i,j], 0))  #Relu
C=te.compute((2,4),lambda i,j:B[2*i,j]) #Strided_slice
D = te.compute((4,2), lambda i,j:C[j,i])  # Transpose
```

$$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$$

$$\begin{bmatrix}2 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$$

$$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$$

TA=B



$$f_{i+1,i}(\vec{v_i}) = f_{i+1}(f_i(\vec{v_i})) = M_{i+1}\times(M_i+\vec{c_i})+\overrightarrow{c_{i+1}}$$

Affine transformation

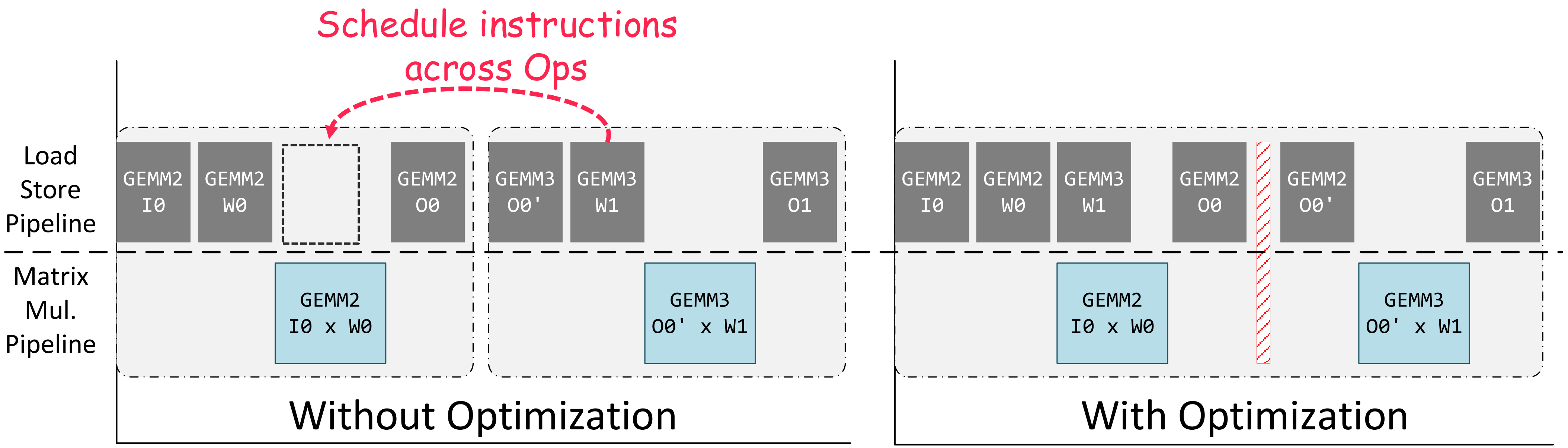
$$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\left(\begin{bmatrix}2 & 0\\0 & 1\end{bmatrix}\left(\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}\right)+\begin{bmatrix}0\\0\end{bmatrix}\right)+\begin{bmatrix}0\\0\end{bmatrix}\Rightarrow\begin{bmatrix}0 & 1\\2 & 0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}+\begin{bmatrix}0\\0\end{bmatrix}$$

```
# Transformed TE
D = te.compute((4,2), lambda i,j:
  tir.if_then_else(A[j,2*i]>0, A[j,2*i], 0))
```

Automatic transformation

Our approach

- Post optimization



Experimental Setup

- Software: Implementation based on TVM 0.8
- Hardware: NVIDIA A100 GPU, CUDA11.7
- Strong Baselines
  - Ansor: we based on Ansor to generate code
  - TensorRT: Vendor optimized compilers
  - Rammer： Microsoft optimized compiler OSDI'20
  - Apollo: MindSpore compilerMLSys'22
  - XLA： JIT compiler in TensorFlow
  - IREE: MLIR based DNN compiler

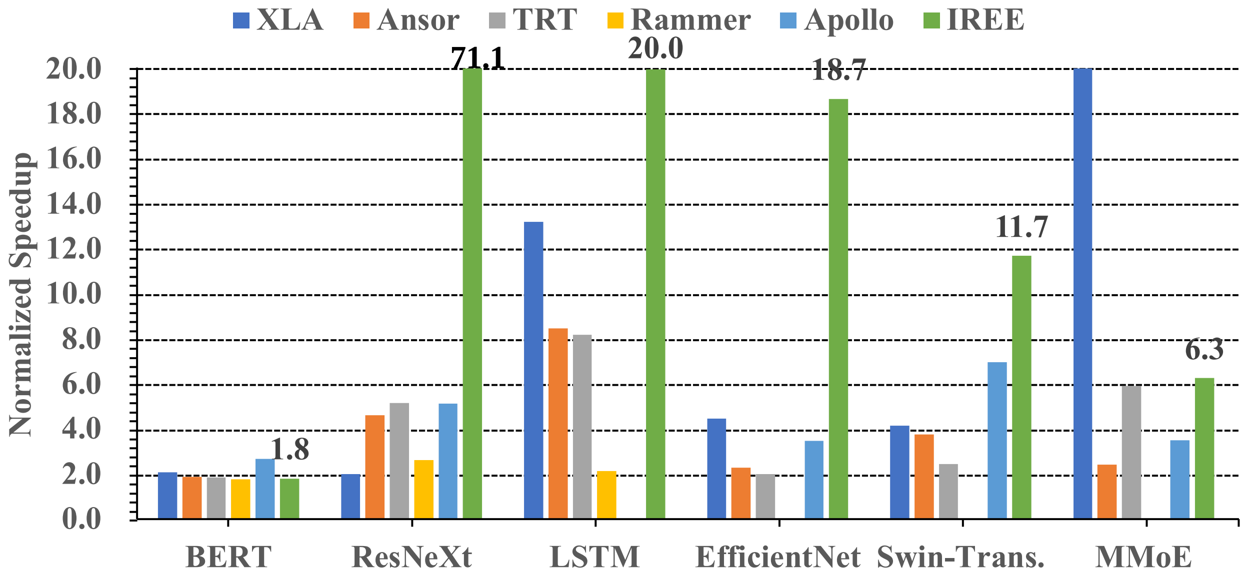
Experimental Setup

- Models

Model	Dataset	Parameters
ResNeXt	ImageNet	#layers:101, bottleneck width: 64d
EfficientNet	ImageNet	Efficient-b0 from the source publication
Swin-Transformer	ImageNet	Base version, patch: 4 and window size: 7
BERT	SQuAD	Base version with 12 layers from TensorRT
LSTM	synthetic	input length: 100, hidden size: 256, layer: 10
MMoE	synthetic	We use the base model

Experimental Results

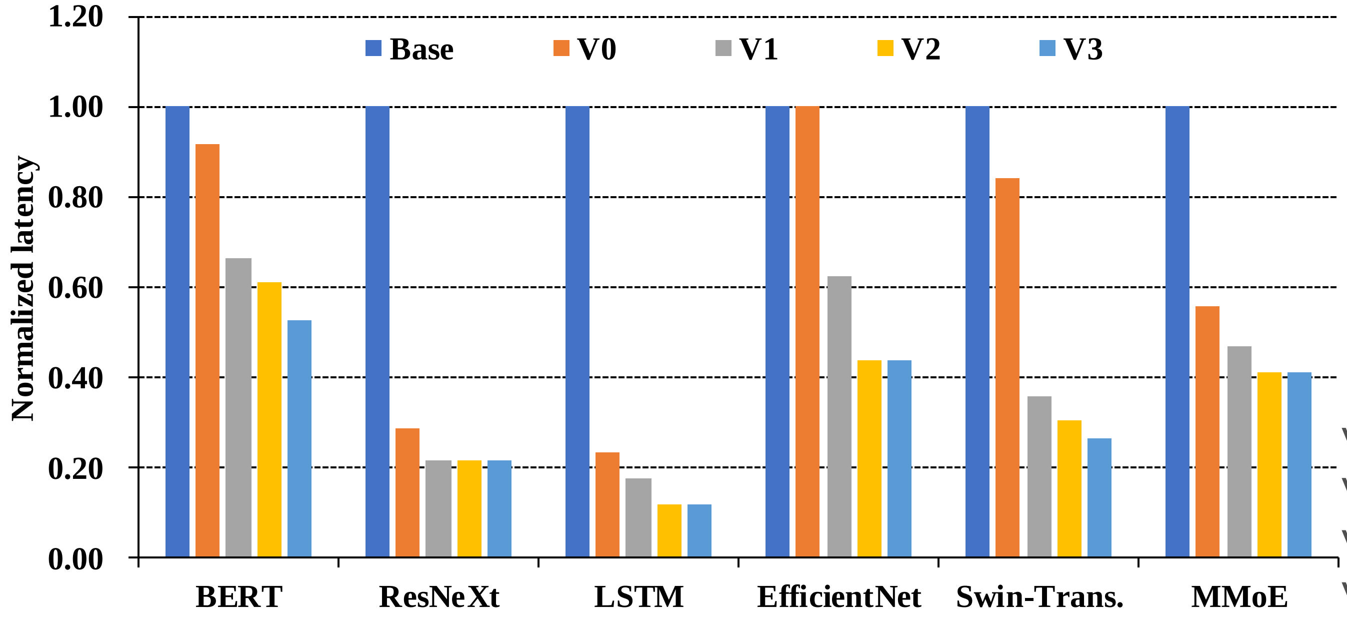
- End-to-end latency
  - 3.94× on average (maximum 8.5×) over Ansor
  - 4.0× g-mean speedup (maximum 7.9×) over XLA



Our compiler significantly outperforms STOA works

Experimental Results

- Performance breakdown
  - Enable each optimization one-by-one



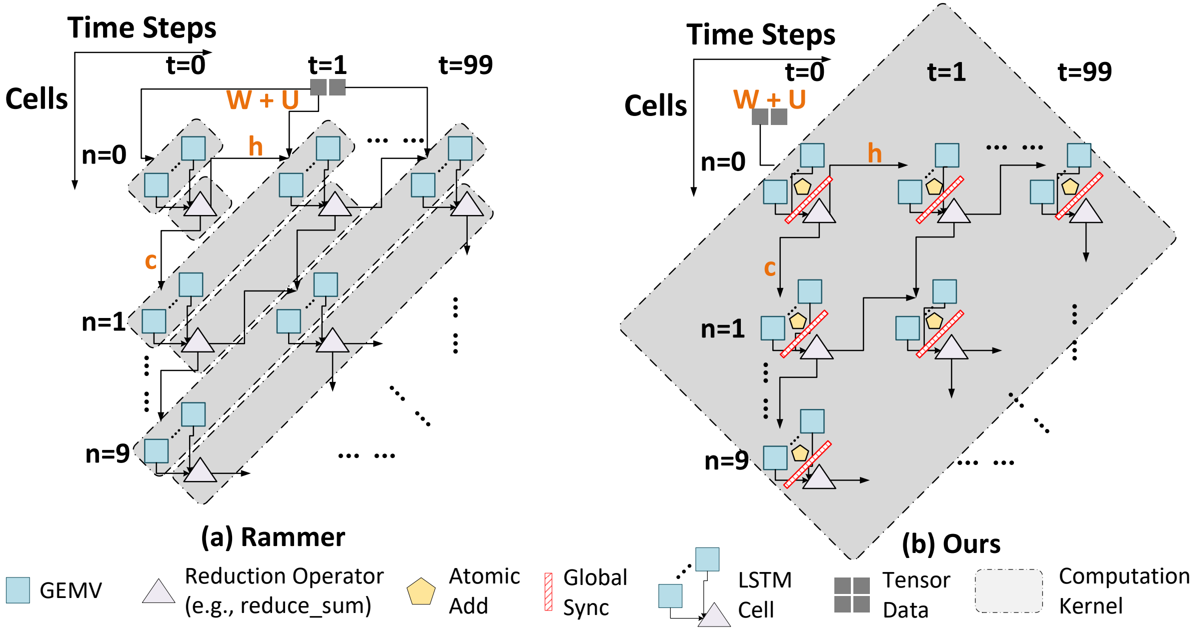
V0: Horizontal trans.
V1: vertical trans.
V2: global sync;
V3: subprogram-level opt.

Each optimization can effectively reduce the latency

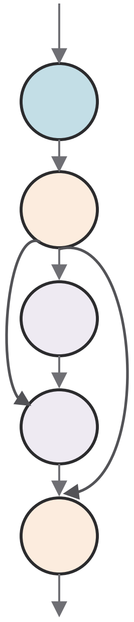
Experimental Results

- Case study on LSTM
  - 10 layers 100 timesteps
  - Rammer 220 vs Our 1 kernel
  - Rammer 1.72ms vs Ours 0.80ms

Metrics	Rammer	Souffle
Dram bytes from global	1911.0MB	21.11MB
Pipeline Utilization (LSU)	20.2%	35.4%
Pipeline Utilization (FMA)	8.0%	19.0%



Conclusion



Lowering Op to TE

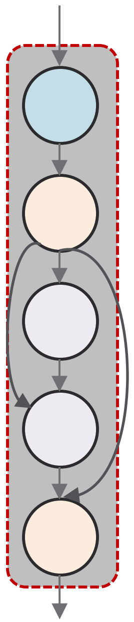
Analysis

```
     rk   = te.reduce_axis((0, 64),)
TE0: O0 = te.compute((64,64), lambda i, j: te.sum(I0[i,rk]*W0[rk,j]),axis=[rk])
TE1: O1 = te.compute((64,64), lambda i, j: te.sigmoid(O0[i, j]))
TE2: O2 = te.compute((64,64), lambda i, j: te.sum(O1[i,rk]*W2[rk,j]),axis=[rk])
TE3: O3 = te.compute((64,64), lambda i, j: O0[i,j] + O2[i,j])
TE4: O4 = te.compute((64,256),lambda i, j: te.sum(O4[i,rk]*W4[rk,j]),axis=[rk])
```

Tensor Expression Program

Transformation

Schedule & Codegen



1 kernel

- Tensor Expression As the Intermedia Representation
- Try to generate the whole model as a single kernel

Top-Down Global Opt.

Q&A